COMPUTER SYSTEM DESCRIPTION LANGUAGE–CSDL

A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
MASTER OF TECHNOLOGY

By

ANANT M. KURKURE

to the

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY KANPUR
AUGUST, 1976

EE-1976-M-KUR-COM

CERTIFICATE

This is to certify that the thesis entitled, 'COMPUTER SYSTEM DESCRIPTION LANGUAGE', is a record of the work carried out under my supervision and that it has not been submitted elsewhere for a degree.

[illegible]

V. K. Vaishnavi
Lecturer in Computer Sciences,
Indian Institute of Technology, Kanpur.

Kanpur
August, 1976.

-: *TO MY LOVING MOTHER*:-

ACKNOWLEDGEMENTS

I would like to express my deep sense of gratitude to my thesis supervisor, Dr. V.K. Vaishnavi, for his sustained interest in the work, constant encouragement, guidance and kind help in innumerable ways. He always devoted his invaluable time to this work for long sessions of discussions.

My regards are due to Dr. V. Rajaraman, Dr. H. V. Sahasrabuddhe, and Dr. K.P.R. Prabhu for their helpful suggestions and guidance.

The assistance given by the Staff, Computer Centre, I.I.T. Kanpur, is gratefully acknowledged.

I also take the opportunity to express my sincere gratitude towards my friends for the pleasant time I had while working here.

My thanks are also due to Mr. S.S. Pethkar for the diligent typing of the thesis in a short time.

Kanpur
August , 1976.

– A. M. Kurkure

COMPUTER SYSTEM DESCRIPTION LANGUAGE - CSDL

ABSTRACT

The need and purposes of a high level language to describe computer system behaviour at program level are discussed. Currently available computer description languages are briefly surveyed. A Computer System Description Language - CSDL is proposed for documenting computer system's behaviour at program level. The language can be used as an input language to a systems simulator which simulates the system at instruction level. Computer systems TDC-316, IBM-1800 and IBM 7044 are described using CSDL.

- * -

CONTENTS

- * -

CHAPTER ONE

INTRODUCTION

A computer system can be described at circuit level, logic level (includes Register Transfer sublevel), program level and PMS level. (2, 3) . Circuit and logic levels except Register Transfer sublevel are well defined. Lot of current research is going on to standardise RT level of description. However even RT level is not suitable to describe large digital systems like computers. Therefore need arises to have a description language at a high level. Once such a high level language to describe the behaviour (1) of computer systems is developed then any computer system can be described at the behavioural level uniformly. Then it will be easy to learn different computer systems described in such a language and will also help in keeping track of old machines, which will be useful while designing a new system. Simulation of any computer system will also become easy, once the interpreter for such a language is available.

A Computer System Description Language (CSDL) is proposed in this thesis. The language is capable of describing the behaviour of the computer system at program level. Example systems TDC 316, IBM 1800 and IBM 7044 are described in the appendices using CSDL.

1.1 Purposes of the Computer System Description Language

There were three purposes kept in mind while developing CSDL. The language should serve as

(a) Documentation language for computer systems.

(b) Input language to a generalised simulator.

(c) Teaching aid.

(a) CSDL as a documentation language for computer systems :

As described in the above paragraphs CSDL can describe the behaviour of computer systems. As it is only behavioural description, timing and concurrency need not be taken into account. Necessary comments to make the description more clear can be added using braces. To describe the behaviour of computer systems, all their basic instructions must be described. Thus CSDL description will be compact and concise. It will help in understanding the system in a short duration of time.

(b) Input Language to generalised simulator (Systems Simulator)

CSDL description will be equivalent to a program. If CSDL interpreter is available then any computer system behaviour at instruction level (22, 30) can be simulated. This will help in developing software for any proposed computer system whose specifications are laid down. Thus the complete debugged software for the proposed computer system will be available at the same time when its hardware is ready. This will avoid in writing a simulator program for every individual machine for the development of the software, and thus will save time. This will be very important application in the sense that software costs about sixty percent of the total cost of the system.

(c) Teaching aid :

When an interpreter described in (b) above is available then it will also be helpful as a teaching aid in a course like Software Engineering. The student can himself describe the system of his interest in CSDL and simulate it on existing computer using CSDL interpreter. Then he can have an experience of developing the software starting from "Boot-strap technique" (22, 30).

1.2 Requirements of the Computer System Description Language :

The various desirable features the language should have are described below :

(1) The language should be useful for documenting the existing and proposed computer systems' behaviour at program level. The notation should be useful as a conveyer among the human beings. It should be precise, concise and elegant. The language should be easy to learn and remember so that the documented computer system could be understood easily. Additional readability can be achieved by adding necessary comments. The whole description should look like a homogeneous program.

(2) To achieve simplicity and familarity, the language should have few primitive programming concepts and they should be used consistently throughout the description.

(3) The language should be a general one. It should be independent of any machine organization, hardware technology etc. , because all of these factors do keep on changing.

(4) The language should be procedural (1) and syntactically simple so that implementing it will become easy.

(5) The language should be similar to other programming languages like FORTRAN IV, ALGOL, PL/I etc. It should be at program level.

(6) The language should have terminology parallel to hardware terminology.

1.3 Objectives and Outline of the thesis :

The objective of this thesis is :

To come up with a procedural computer description language to describe the behaviour of computer systems at the programming level, which could also be used as an input language to a generalized simulator (CSDL interpreter) to simulate the described computer system at instruction level.

Outline of the thesis :

Chapter 2 : This chapter deals with various levels of system description. A brief survey of the existing languages at programming and higher levels is presented.

Chapter 3 : This chapter defines and describes the Computer System Description Language. It also includes its syntax specifications.

Chapter 4 : This chapter explains in brief the various examples of system description given in the appendices and justifies the purposes, objectives and requirements that are set to in this chapter.

Chapter 5 : This chapter concludes the thesis and a few suggestions for further work are discussed.

—*—

CHAPTER TWO

A SURVEY OF SYSTEM DESCRIPTION LEVELS AND LANGUAGES

Computer system description can be classified as behavioural, functional or structural description (1) . Another way of classification given by Bell and Newell (2, 3) is PMS, program, logic and circuit level description. The languages describing the systems can be procedural or nonprocedural (1). A brief survey of these areas is presented here. Further some of the high level languages are also presented discussing their various features. (1,2,3,7,11,21,25 etc.)

2.1 Behavioural, Functional and Structural Descriptions:

In a system description, several levels of details can be used. They range from behavioural description (in which properties of the system are specified in terms of the input/output relationship between variables - a black box approach)to structural descriptions (where the system is described in terms of the real hardware components and their interconnections). Intermediate functional descriptions represent the system in terms of the actual components and their functional relationship or algorithm.

Behavioural descriptions are closer to the conventional programs in most programming languages. Complex expressions and operations are allowed for simplicity in the description. Variables and operators donot necessarily have a hardware counterpart, and timing details are ignored. Concurrency is also not taken into account because the only purpose is to describe the behaviour of the system. Work done so far for this level is very limited. This is equivalent to program level discussed in the next section.

Functional descriptions are closer to the real hardware. They describe the system as an algorithm in terms of the real registers components of the machine. The operators may or may not be hardware primitives and expressions can be of complicated nature. Timing and concurrency is taken into account. The ISP notation suggested by Bell and Newell falls into this type.

Structural descriptions represent the system in terms of the hardware components. Operators have hardware counterparts (i.e. they are primitives) and the descriptions tend to be more detail than the other two levels. Being closer to physical implementation, timing is described in terms of clock pulses or event completed signals. All register transfer languages fall into this category.

2.2 Various Levels of Description :

Alternatively a system can be completely described at any one of the four levels,viz. circuit, logic, program and PMS, if the set of elements constituting the system, the interconnections and the information transfer between elements are specified. Each level of description has certain set of primitives or basic elements. Generally primitives in one level are subsystems in the lower level. Logic level can be further divided as switching circuit level and register transfer level. Lower levels upto switching circuit level are standardised. Register transfer level is expected to get standardised within next few years. Here only levels above register transfer level are discussed.

2.2.1 Register Transfer Level

The components of this level are registers, clock, data operators and functional transfers between registers. Sometimes arithmetic circuits are also used as elements. Subsystems are described by both combinatorial logic networks and registers. The blocks and subsystems can be used to obtain other complex blocks and subsystems. The information transfer from one register to other register can be simple with or without modification. The transfers are activated by control signals and clock pulses. There are many languages proposed at this level and a few of them have been implemented successfully (1,7,11,12,13 ,20,21,23,27,29, 31). Languages at this level give the structural description of the system. A brief survey of register transfer languages is presented in the next section.

2.2.2 : Program Level :

This is not only a unique level of description for digital technology (as is the logic level) but is uniquely associated with digital computers, namely, with those digital devices that have a central component that interprets a programming language. There are many uses of digital technology in instrumentation and controls which donot require such an interpretation device and hence have a logic level but no program level. Thus program level is uniquely associated with digital computers only.

Components of program level are memory cells, instructions, operators, controls, and interpreter. The memories hold data structures which represent things both inside and outside the memory. The operations take various data structures as inputs and produce new data structures, which again reside in memories. Thus behaviour of the system is the time pattern of data structures held in its memories. The unique feature of the program level is the representation it provides for combining components, that is, for specifying what operations are to be executed on what data structures. Each instruction specifies that a given operation (or operations) be executed on specified data structures. Superimposed on this is a control structure that specifies which instruction is to be interpreted next . Normally this is done in the order in which the instructions are given, with jumps out of sequence specified by branch instructions.

There is no unique representation for programming level. Each computer has its own instruction set that is different from that of other computers. Bell and Newell (2,3) developed a new notation called ISP (Instruction Set Processor) which is described in brief in the next section, to provide a uniform way of describing instruction set of the existing computers and also that of the future. They tried to develop it at program level but it gives the description somewhere in between program and register transfer level. ISP description is a functional description of the system. However it has been extended to describe computational expressions as in high level languages like FORTRAN and ALGOL.

Program level description of a system is supposed to give its behavioural description. Once a language for describing a computer system at this level is available then we can write an interpreter for it. This will facilitate in simulating the behaviour of the computer system.

2.2.3 PMS Level (Processor Memory Switch)

This is the topmost level of describing computer systems. At PMS level (2,3) computer systems can be described in terms of primitives like processors, memories, switches, transducers, controls, links, data operators and peripherals. Such a description of a system is diagramatic. Information flow is indicated by making use of thick lines. Using such a primitive set, the components like computers and computer networks evolve out. This level exists informally. It is the view taken of a computer system when its most aggregate behaviour is considered. It is possible to evaluate the gross properties of a computer system at this level. There is substantial amount of work done in simulation and design of systems which incorporate multiprocessing, multiprogramming, time sharing with large I/O devices, computer networks etc., at this level. The operating characteristics of the primitives are costs, memory capabilities, information flow rates, power etc. There is no uniform language at this level and even no standard name as such.

2.3 Procedural and Non-Procedural Languages :

Any computer system description language can be either procedural or non-procedural (1). Non-procedural languages tend to impose many restrictions on the user. The sequence of operations (the algorithms) must be described by providing the timing and the conditions to execute the operations. This kind of detail is irrelevent if the designer only wants to describe algorithms in terms of the I/O sequences, without any consideration to the actual clock pulses, or state register values. The examples of such kind are CDL (7), DDL (11), DSDL(21) etc.

Procedural languages are better suited for behavioural descriptions. The algorithm is described as a sequence of steps as in a conventional programming languages and the details about the timing and the conditions can be ignored. Procedural languages are capable of describing the system using conditional statements. The conditions are labels used in nonprocedural languages. Examples of the procedural languages are FORTRAN-like language proposed by Metze and Seshu (25) and ISP (2,3).

With all the above background, the objectives of the thesis are clear. Now we proceed to the next section in which some high level languages are briefly surveyed.

2.4 High Level Computer System Description Languages :

This section is divided into three subsections viz. RT languages, languages similar to conventional programming languages and ISP notation. Each of them are briefly presented in the following pages.

2.4.1 RT- Languages :

There are so many RT languages proposed out of which a few have been successfully implemented and are being used in the design of digital systems. Examples are CDL (7), DDL (11), DSDL (21), AHPL (17), LOCS (34) etc. They are all nonprocedural languages They describe the system at structural level. All of them are developed for design purpose, i.e. design automation (6), and not for description (documentation) purpose. These languages are not powerful enough to describe the larger digital systems like computers in a concise and precise manner. The description becomes too lengthy. While describing a digital system, register transfer languages descend down one level below to describe particular pieces of hardware. Thus the languages are not purely at RT level.

2.4.2 Languages similar to conventional programming languages :

There have been a few proposals to describe the computer system at its structural level using the languages similar to other conventional programming languages like FORTRAN, ALGOL, PL/I etc. But these languages are not suitable for hardware systems because of their special nature. There have also been some efforts towards using FORTRAN as a simulation language for digital systems. ALGOL and PL/I provide facilities like block structure feature and labelling which donot exist in FORTRAN. All of these languages are useful to a limited extent for describing digital system at its behavioural or structural level because these languages are developed for computational purposes. Moreover they donot have terminology used in digital systems. Again the facility to deal with parallel operations, timing and control signals, and sequencing information is totally non-existent which is useful in design simulation.

A FORTRAN like computer design and description language is suggested by Seshu and Metze (25). The language consists of two parts : global description and control part. In digital systems subcontrols can operate in parallel such subcontrols which operate in parallel are declared in global description. Some constants are also defined in global part, which can be used in the other parts of the description synonymously, so that just by changing the global description one can alter the design keeping other parts of the program same. The global description also contains design contrainsts like cost, speed etc. Control part is further subdivided into main control and subcontrols, similar to main program and subroutines in Fortran. Subcontrol represents a hardware block. Examples are subcontrol ARITH, subcontrol DECODE etc., which reflect arithmetic and decoder hardware counterparts respectively. Systems library approach is introduced to help in design of digital systems. They have suggested writing systems compiler, for such an input language, which will translate it into some intermediate language. This is hardware independent part. Further the intermediate language output of the systems compiler according to them can be translated to implementation details using systems library approach, by logic compiler. The main purpose of the language has been to automate the design process. They tried to do it but without success. A description (program) written according to this proposal of Sheshu and Metze is self documenting and it reflects the machine organization. It is not, however, easy to understand such a system description which is a high level structural description. To have an automated system of design, as proposed by them , it is necessary to have a rich systems library which contains all types of designs of component digital systems. It is not easy to have such a library. Again, the whole design can be asynchronous only.

2.4.3 . Instruction-Set-Processor (ISP) :

The ISP descriptive scheme suggested by Bell and Newell (2,3) gives the informal description of the computer system. It is meant to provide a uniform way of describing instruction sets at functional level of description. It takes concurrency into account by using symbols "next" and"; " . Expressions can be complicated. Again timing details are ignored. Thus it is certainly above RT level. At program level sequencing is achieved by executing instruction by instruction with the jumps out of the sequence by branch instructions or procedure calls. In ISP it is achieved by "next" and ";" allowing concurrent operations. Thus ISP is not at program level. Therefore we can say that it lies in between Program and RT level. An interpretor for ISP as such can not be implemented. It is necessary to investigate some simulation language which will put ISP as an operational tool. It is only functional description of the system. But to understand these ISP descriptions of the computer systems, it is necessary to get familiar with various symbols used and various rules of descriptions. It is not similar to other conventional programming languages; moreover it is notational. Bell and Newell have described many existing computer systems using ISP. It is useful as a description language. But it is necessary that the notation ISP must become formal programming language so that analysis and synthesis procedures can be carried on automatically.

In brief the description is like this :

The instruction set of the computer is described by using instruction expressions which have the form :

$$< \text{condition} > \quad \rightarrow \quad < \text{action sequence} >$$

The condition describes when the instruction will be evoked, and the action sequence describes what transformations of data takes place between what memories. The operator ($\rightarrow$) is control operator.

Modification of memory bits in memory cell has the form :

$$< \text{memory-expression} > \quad \leftarrow \quad < \text{data expression} >$$

The operator ($\leftarrow$) is transmit operator. Left side describes the memory location where the right side (data-expression) is stored.

With this background and objectives and requirements of the high level language in mind, we are in a position to develop such a language (Computer System Description Language- CSDL) which is described in the next chapter.

- * -

CHAPTER THREE

COMPUTER SYSTEM DESCRIPTION LANGUAGE

The Computer System Description Language described in this chapter is capable of describing the behaviour of the system at program level. The computer systems TDC-316, IBM 1800 and IBM 7044 are described in CSDL in appendices I, II and III respectively. In the first section of this chapter CSDL is described informally and in the second section its syntax specifications are given using BNF notation (16).

3.1 The Language

Any CSDL program or a description of the computer system has the following format.

(1)	System declaration	Heading
(2)	Memory elements declaration	Body or Block
(3)	Procedure declaration	
(4)	Statement part	

Each of them is described in details below.

Character set :

The language has the following sets of characters.

(1) Letters : All capital letters A to Z

(2) Digits : All decimal digits 0 to 9

(3) Special characters :

+ - / *) (≠ = } {] [∧ ∨ ¬ ↑ <

, ↺ ↻ ⤓ ↥ ↤ ↦ ; > : °

↓ ⊕

3.1.1 System declaration :

This is the heading of the program. System declaration gives the identification of the system to be described

Example,

SYSTEM TDC-316

Here SYSTEM is the key word and TDC-2316 is the system to be described.

3.1.2 Memory elements declaration :

This part consists of Register declaration, Memory declaration, Array and Equivalent declarations, and Integer declaration. All processor registers, primary memory and integer variables that are referenced to in the following parts of the program are defined under this declaration. This is similar to that in the language Pascal (33) .

(a) Register declaration :

A register is an ordered set of flipflops and each stores one bit of information. Flipflops and switches can be considered as one bit registers. Register'n' bits in length can be represented as R(0:n-1) or R(n-1:0) or R(1:n) according to the convention used. Integer prior to ":" represents Most Significant Bit (MSB), whereas integer following ":" represents Least Significant Bit (LSB).

Register arrays are represented as XR(1:3;0:15). Here part prior to ";" indicates dimension of the array XR and part following ";" is the size of each element of the array (similar to above example R(0:n-1).). Single bit registers (flipflops and switches) are represented by means of identifiers only.

Example,

```
REGISTER          A(0:15),      XR(1:3;0:15),S
```

Here Accumulator A, array XR and switch S are declared as registers. If there are more than one declaration then they are separated by commas.

In addition, the language provides another way of representation or combination of both. The above declaration XR(1:3;0:15) can also be represented as XR(1,2,3;0:15). This facilitates in declarations like XR(1,2,4; 0:15), AC(Q,P,1:35) etc. Here index registers 1,2 and 4 and accumulator of 37 bits (Q,P and 1 to 35) are declared.

Part of register can be defined as a subregister and this facilitates frequent reference to these selected sequences of bits. Subregisters donot exist at all as separate hardware registers. It is only convenient naming for frequent referencing. The use of a subregister makes it easier to identify a part of a register which usually provides a meaningful indication of the function, that the part of register performs.

Example,

```
SUBREGISTER       OP(0:4) = CR(0:4),
                  TAG(0:1) =CR(5:6).
```

Here OP and TAG are subregisters declared under the key SUBREGISTER. Right hand side of the declaration is the identifier declared earlier in REGISTER declaration. Left part and right part have got the same syntax as explained above.

Concatenation register gives a name to the concatenation of two or more registers or subregisters declared earlier. This also does not exist as a separate hardware register but provided for convenience while describing the system.

Example,

```
REGISTER   A(0:15), Q(0:15)
CONCAT  AQ(0:31) = A°Q
```

Here '°' is the concatenation operator and AQ is the single 32 bit register formed by A and Q. Again both right and left parts of the declaration have similar syntax as explained above.

(b) <u>Memory declaration</u> :

A memory in a digital computer system can be regarded as an array of registers capable of storing information which can be accessed. The registers are offen regarded to as the locations of the memory. The address register contains the memory location address to be read or written as the case may be.

Memory declaration is similar to register array declaration. Similar to SUBREGISTER declaration, SUBMEMORY declaration is also there. In this declaration part syntax is same.

Example,

```
    MEMORY  M(0:32768; 15:0)
SUBMEMORY M1(0:8191;15:0) = M(0:8191;15:0)
          M2(0:8191;15:0) = M(8192:16383;15:0) etc.
```

In the above example, 'M' is the main memory, whereas M1 and M2 are submemories or memory modules. Here part prior to ";" gives the capacity of the memory in number of words and following part gives the memory word format.

(c) Arrays and Equivalent declaration :

In some computer systems part of core memory is used for processor registers. In such cases processor registers are firstly declared under ARRAY declaration if they are in arrays. And then under Equivalent declaration each element of the array and other processor registers if they are there, are equated to actual corresponding memory locations as explained below :

Example,

```
Memory   M(0:32767;15:0)
ARRAY  GPR(1:7;15:0)
EQUIVALENT  GPR(1) = M(32752),
            GPR(2) = M(32753) and so on.
            PC = M(32762) etc.
```

In above example, memory location 32,752 is taken as GPR(1), 32,753 as GPR(2) and so on. Processor register PC is memory location 32,762.

(d) Integer declaration :

In this declaration part integer variables used in the program are defined.

Example,

```
INTEGER    Z, ZZ, Z1 .
```

3.1.3 Procedure Declaration :

Program body consists of memory elements declaration, procedure declaration and statement part. Procedure may or may not have memory elements declaration part. But statement part is compulsory for a procedure. Procedure can also call another procedure. The memory elements declared in main body are global to any procedure declared in it, and those declared in a procedure are local to it and can be referenced within a scope of that procedure only.

Example,

```
PROCEDURE      EFFADDR(Z)
```

This procedure is declared to calculate effective address.

3.1.4 Statement Part :

(a) Basic Operators :

The function to be carried out by a logic network during one or more clock period is represented by a symbol which corresponds to an operator. Basic operators are those that are very frequently used. They can be classified as logical, functional and arithmetic. The following table gives the operators, symbols used in description and the type of the operation.

	Operator	Symbol	Type
Logical			
	NOT	$\neg$	Unary
	OR	$\vee$	Binary
	AND	$\wedge$	"
	Exclusive-OR	$\oplus$	"
	NOR	$\downarrow$	"
	NAND	$\uparrow$	"

	Operator	Symbol	Type
Functional			
	Shift Left	$\leftarrow\!\perp$	Unary
	Shift Right	$\perp\!\rightarrow$	"
	Count Up	$\underline{\uparrow}$	"
	Count Down	$\underline{\downarrow}$	"
	Circulate Left	↺	"
	Circulate Right	↻	"
ARITHMETIC			
	Addition	+	Binary
	subtraction	-	"
	Multiplication	*	"
	Division	/	"

(b) Expressions and Statements :

Using above operations in combination we can get logical or arithmetic expressions. Syntax of statement is explained in next section Use of parenthesis can be made to avoid ambiguity while forming the expressions. Thus complicated expressions are combination of these basic operations. Logical or arithmetic statements can be obtained by equating an identifier to the logical or arithmetic expressions respectively. This is equivalent to register transfers with or without any modification.. Similarly we can intialize registers which are already declared. For example, A=0. This will initialize register A to all zeros.

Relational expressions (rather conditions) are formed by two identifiers separated by relational operators like $>$, $<$, $=$ and $\neq$.

(c) Compound Statement :

The compound statement specifies that its component statements be executed in the same sequence as they are written. The symbols BEGIN and END act as statement brackets .

Exampl e,

```
IF(¬SKIPCOND) THEN   BEGIN
                     I=Z+1
                     M(Z)=I
                     END
```

Here if SKIPCOND=0 then BEGIN...END block is executed otherwise control is transfered to next sequential instruction. The block includes two simple statements I = Z+1 and M(Z) = I which are executed serially.

(d) Control Statements :

Some familiar control statements are included in CSDL. These statements enable to describe the behaviour of the system at program level. It is one of the primitives for program level. They are explained below.

(I) Conditional Statements:

A conditional Statement, IF or CASE (decoder operation) statement, selects single statement of its component statements for execution. The IF statement specifies that a statement be executed only if a certain condition is true, if false then either no statement (in this case control is transfered to next sequential statement) or the statement following the symbol ELSE is executed. For logical conditions, the convention is, if the value of the logical expression is 1 then it is true, otherwise false.

The two forms of IF statement are :

```
  IF <Condition > THEN  <Statement>
& IF <CONDITION> THEN  <Statement>
                 ELSE  <Statement>
```

The <Condition> can be logical or relational expression. If <Statement> is combination of more than one statement then it is <compound statement>. <Statement> can be a procedure call.

Examples.

```
  (i)  IF (RUN)  THEN CR(0:1) = M(I)°M(I+1)
 (ii)  IF (F)    THEN I= I+2  ELSE I = I+1
(iii)  IF (¬IA)  THEN Z= ADDR + XR(T)
                 ELSE    BEGIN
                         Z1 = ADDR + XR(T)
                         Z  = M(Z1)
                         END
 (iv)  IF (CR(9)=1) THEN  BROINT
```

In example (iv) BROINT is procedure call. In first three examples conditions are logical and in fourth one condition is relational.

CASE Statement :

The case statement consists of an expression (Selector) and a list of component statements, each being labelled by a constant of the tyne of the type of the selector. In CSDL it is integer represented in Octal. It selects for execution the statement whose label is equal to the current value of the selector. This is equivalent to decoder GO TO statement in FORTRAN IV. Upon completion of that statement, control is transfered to the end of the CASE statement i.e. CASEND.

The form of the statement is :

```
CASE   <expression>  OF
    <Case-label list> :  <Statement>
    <Case-label list> :  <Statement>
     ..............................
     ..............................
    <Case-label list> :  <Statement>
CASEND
```

Here <Statement> is compound statement.

Example,

```
CASE   OP   OF
 30  :  BEGIN
        A= M(Z)
        END
...............
...............
...............
32 :  BEGIN
      AQ = A*M(Z)
      END
CASEND
```

(II) Iterative Statements :

Some cases like interpreter cycle, shift process etc. are repeated more than one time while describing the behaviour of the system. Repetitions are continued until certain condition is satisfied or for fixed number of times. Such a repetative execution of sequence of statements for a definite number of times is achieved by introducing two statements of the following type.

(i) REPEAT .... UNTIL ..... Statement :

The form of the statement is :

REPEAT ⟨Compound Statement⟩ UNTIL ⟨Condition⟩

The sequence of statements consituting compound statement is executed until the condition is true. It stops iterating as soon as condition becomes false.

Example,

```
REPEAT   BEGIN
         IR = M(PC)
         INSTRXEQ
         PC = PC+2
         END
UNTIL    (STOP= 0)
```

Above example represents interpretation process. As soon as STOP becomes 1 it stops iterating.

(ii) FOR statement :

The form of the statement is :

FOR ⟨Control variable⟩ : = ⟨initial value⟩ To ⟨final value⟩ DO
⟨Compound Statement⟩

This statement indicates that a statement be repeatedly executed while a progression of values is assigned to the control variable of the FOR statement.

Example,

```
FOR   K=1  TO  8  DO
      BEGIN
        PB = PB +  A(1)
        ↵ A
      END
```

⟨final value⟩ can be integer variable or register value declared earlier.

(e) Procedure Calls :

This is equivalent to procedure in the language Pascal(33). Procedure itself may call other procedure provided the called procedure is not in the domain of calling procedure. When there is a call to a particular procedure, the control is transfered to that procedure. After completing execution of the called procedure, control is again transfered back to the next sequential statement.

Example,

```
EFFADDR(Z)
```

Execution of this statement causes the transfer of control to EFFADDR procedure. After completion of execution of EFFADDR procedure, control is transfered back to the next statement of the calling procedure.

(f) Input-Output :

Two input-output statements are included. One is READ statement which reads the card from the indicated device into the given memory location. Other one is WRITE statement, which writes on the indicated device from the given memory location.

Examples,

```
READ  <device addr.>  , M(Z)
WRITE <device addr.>  , M(Z)
```

Timing and concurrency are not taken into account as the language CSDL is only for behavioural description of the computer systems at program level.

4.2 SYNTAX OF THE LANGUAGE :

It is essential for any language to have concise and compact syntactic specifications, i.e. grammer. For the production or recognition of the allowable strings in the language rules can be formulated according to its specifications. For this purpose Backus Naur Form (BNF) notation is used to specify the syntactic specification of the CSDL. (16) .

4 2.1 Basic Elements :

The basic elements of the syntax specification are terminal and nonterminal (also called as syntactic entities) symbols. The terminal symbols are grouped into three categories viz. letters, digits and special characters, In BNF they can be represented as:

<letter > : : = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|

< digit > : : = 0|1|2|3|4|5|6|7|8|9

< Special Character > : : = + | - | / | * |) | (| . | = | ≠ | } | { | [|] | ∧ | ∨ | ⊣ | ↤ | ↦ | ⊥ | ⊥ | ⊕ | ⊘ | ○ | ↓ | ↑ | ; | , | > | < | ° | :

The basic building blocks of the language are identifiers and integer constants. Their syntax is :

<identifier > : : = < letter >| <identifier> | <letter> |< identifier > <digit >

<integer > : : = < digit >|< integer > < digit >

4.2.2 System declaration :

< System declaration> : : = SYSTEM <identifier>

Here onwards square brackets will be used to indicate that the contents are optional while braces will be used to indicate repetitions of the contents as required.

4.2.3 Memory elements declaration :

(a) Register declaration :

```
<register declaration> ::= REGISTER <name 1> {,<name 1> }

        <name 1)> ::  =<identifier> (<name 2> ;<name 2> ) | <identifier>
                    [{<identifier> ,}]                      [{,<identifier>}]
      <name 2> ::=  [{      or      }] <integer> : <integer>[{      or      }]
                    [{<integer> ,   }]                      [{,<integer>    }]

                 | <identifier>   {      <identifier> }
                 |     or        ,{          or       }
                 |  <integer>     {      <integer>    }

 <subregister declaration> ::=  SUBREGISTER <name 1> = <name 1>
                                          {  <name 1> =< name 1>}

 < concat register declaration> :: = CONCATREG< name 1> =< name 1> °<name1 >


                                        {,< name 1> = <name  1> ° <name1 >}
```

(b) Memory declaration :

```
 < memory declaration> ::  MEMORY < name 3>

     < name 3> ::= <identifier>  (< name  2> ;< name 2> )

 < submemory declaration> ::=  SUBMEMORY  <name 3> =< name  3>
                                           {,< name 3> = <name 3 > }
```

(c) Array and Equivalent declarations:

```
 <array declaration> ::  = ARRAY  <name 3> {, < name  3>}

 <eqvt. declaration> ::  = EQUIVALENT < name  1> = <name  3>
                                {,< name 1> = <name 3>}
```

(d) Integer declaration :

< integer declaration > : : = INTEGER < identifier > { , < identifier > } .

4.2.4 Procedure declaration :

< procedure declaration > : : = . PROCEDURE < proc . identifier > [(< arg.list >)]

< procedure identifier > : : = < identifier >

< arg. list > : : = < identifier > { , < identifier > }

Syntax for procedure call is

< procedure call > : : = <procedure identifier> [(< arg. list >)]

4.2.5 Statement part :

< statement part> : : < compound statement>

< compound statement > : : = BEGIN <statement>{ < statement>}
END

< statement > : : = <compound statement>| < logical statement>|
<arithmetic statement>| <functional statement >|
<control statement> | <set constant statement>

```
                                              ┌ <name 1 > ┐
< logical statement > : : = <name 1 >   =     │    or     │ <logical operator >
                                              └ <name 4 > ┘
                                                                <name 1 >
                                                                   or
                                                                <name 4 >
```

< logical operator > : : = ¬ | ∨ | ∧ | ⊕ | ↑ | ↓

```
                                  < name 1>      < name 1 >
< arithmetic statement> : : =        or      =      or      < arith.operator >
                                  < name 4>      < name 4 >
                                                                < name 1>
                                                                   or
                                                                < name 4>
```

```
<arith. operator>  ::=  + | - | / | *

<functional statement>  ::= <functional operator><name 1>

<functional operator>  ::  ⍅|⍆|⍖|⍏|↺|↻

<set constant statement> ::= <name 1> = <integer>

<control statement> ::= REPEAT <compound statement>
                        UNTIL <condition>|
                        FOR <control variable> = <initial value> TO
                                                 <final value> DO
                                             <compound statement>|
                   <conditional statement>| <I-O statement>

<condition> ::= <logical expression>|<relational expression>

<logical expression> ::= ┌<name 1>┐                        <name 1>
                         │   or   │  <logical operator>     or
                         └<name 4>┘                        <name 4>

                          <name 1>                        <name 1>
<relational expression> ::=  or    <relational operator>     or
                          <name 4>                        <name 4>

<relational operator>   ::= >|< | =| ≠

<control variable>      ::= <identifier>

 <initial value>        ::= <integer>

 <final value>          ::= <integer>
                                                  <statement>
<conditional statement> ::= IF <condition> THEN      or
                                                  <compound statement>|
                            If <condition> THEN   <statement>
                                                     or
                                                  <compound statement>
                                           ELSE   <statement>
                                                     or
                                                  <compound statement>|
                         CASE < name  1 >  OF
                     < case-label list> :< compound statement>
                     -----------------------------------------
                     < case label list> :< compound statement>
                      CASEND
< case label list>  ::=< name4>  | < case label list> < name 4>
     < name 4>      ::=  0|1|2|3|4|5|6|7
< I/O statement>    ::=< operation>  (< device address> ),< name 3>
      < operation>  ::=  READ|WRITE
   < device address>::= < integer> | < identifier>
```

-*-

CHAPTER FOUR

EXAMPLES AND JUSTIFICATIONS

The example systems selected to demonstrate the usefulness of the language CSDL as a documentation language are TDC 316 ,(32) IBM 1800 (18) and IBM 7044(19), which are presented in appendices I, II and III respectively. These describe their behaviour at programming level. Each of these systems is discussed in brief in the following sections.

4.1. TDC-316 -

This is a third generation computer system having integrated circuits technology. It has 28K words core storage (user accessible), which is divided in four modules of 8, 8, 8 and 4K each. Each word is further divided in two bytes of 8 bits each. It is byte addressable. Processor registers like GPR sets, Program counter, Processor status register, stack limit register etc. occupy the part of core storage. (4K) (Thus total 32K core storage). CSDL allows such descriptions by means of its ARRAY AND EQUIVALENT declarations. Memory, Processor, Console, instruction, data and address formats etc. are described using memory elements declarations of CSDL. Then various procedures like effective address calculation, instruction execution etc. are described in procedure declaration part. At the end statement part of the main program i.e. interpretation cycle of TDC-316 follows.

4.2 IBM 1800 :

This is also a third generation computer system. It has 16K words core memory. Each word consists of 18 bits. Here two bits P and S are added as parity and storage protection bits. Memory, Processor, instruction formats, IOCC (input-output control code) format etc. are described in memory elements declaration part. Various procedures like effective address calculation, instruction execution, IOCC execution etc. are described in procedure declaration part. At the end interpretation cycle of IBM 1800 follows.

4.3 IBM 7044 :

This is a second generation computer system. It has 32K words of 36 bits each, core storage. Memory, Processor, console, instruction and data formats etc. are described in memory elements declaration. Various procedures like effective address calculation, instruction execution etc. are described and then at the end interpretation cycle of IBM- 7044 follows.

In all the above systems the general format of description is same. Once one system description is understood completely then it is easy to follow that of others. This shows the consistency in descriptions. The language does not take into account concurrent processes in hardware instead they are represented serially. This is because the language is meant to give only behavioural description of the system. The whole description of the system looks like a single program.

4.4. Justifications :

Looking back to our objectives set before, the language is procedural one. It is capable of describing the behaviour of the computer system at program level. It is used successfully to document the existing computer behaviour at program level. It is precise, concise and elegant. It is easy to understand. The language is similar to other programming languages like Pascal, Algol etc. Therefore it is easy to learn and remember also.While describing the systems addition of necessary comments make the description more readable. The language has got few primitive programming concepts and they are used consistently while describing the systems. This makes the language simple and familiar. It is independent of any machine, organization, hardware technology etc. The language has got the terminology similar to hardware systems. Description of the computer system using CSDL is easily understandable. The language allows memory element declarations of complicated nature. Almost each and every element in memory element declaration is an array. The language has the primitives like memory cells, instructions, operators, controls, and interpreter, which are the primitives for the program level while describing the behaviour of the system. It describes the system with regard to input-output relationships, ignoring intermediate register transfers. Thus it gives behavioural description of the system also. Addition of control statements make it procedural.

As far as powers of CSDL are considered, it is capable of describing any computer system to the extent that possible by ISP description. Moreover adding input-output instructions in the language make it possible to read from or write on any device that is provided for the computer system. This will be useful in its one of the applications i.e. systems simulator, when some data is to be inputed or outputed. The language is closer to ISP. CSDL gives the behavioural description while ISP gives functional description taking concurrency into account. CSDL can be implemented on existing computers whereas ISP as such can not be. Descriptions in ISP are notational while in CSDL they are descriptive to some extent. ISP gives informal description of the system whereas CSDL gives formal description. Any ISP description can be translated into CSDL which will describe the behaviour of the systom.

Though other conventional programming languages like Pascal, Algol etc. can be used to simulate the behaviour of the computer system, they are not suitable for documenting the computer system, because these languages are developed essentially for computation purposes, they do not have terminology used in digital system description, and using these languages description of computer elements and basic operations is not feasible. Whereas CSDL is capable of describing the computer system using terminology similar to that in digital systems and has got the required basic operations set. Moreover, the same description can be used as an input to the generalised simulator which further simulates that system on the source computer. Thus while describing the computer system using CSDL, its simulator is also ready.

- * -

CHAPTER FIVE

CONCLUSIONS

The Computer System Description Language proposed in this thesis is developed for describing the present and future computer systems' behaviour at programming level. The usefulness of the language is demonstrated by describing the present computer systems TDC-316, IBM 1800 and IBM 7044. The language is also useful as an input language to the systems simulator which will simulate the target computer system, described in CSDL, on the source computer at instruction level. Then starting with "Bootstrap Technique" one can develop the software for the target machine. Thus the further step is to write the systems simulator for which input language will be CSDL.

It is significant to state here that there is current research going on in Register Transfer Modules (RTMs).(4) . Efforts are being made to standardise the set of RTMs which can be used to implement any control operation. Thus once the RTMs set is defined, then with some changes in CSDL like concurrency, the language will be useful as design language. The task can then be automated by writing a program which will accept revised CSDL as an input language and give out the computer system implementation details with inter-connections between various modules. (Complete Wiring diagram).

When such an integrated system will be ready then it can be used as a good research tool in design, as a teaching aid and for development of software for any proposed computer as soon as its specifications are laid down.

The language proposed is formal and is like other conventional programming languages. It has familiar control statements like IF.....THEN....., IF.....THEN.....ELSE..... & the CASE statement which is equivalent to decoder operation. It includes compound statement like BEGIN....END. Iterations are handled by use of REPEAT.....UNTIL.....and FOR loops. The language is easy to learn and remember . The descriptions using CSDL are easy to understand.

- * -

REFERENCES

(1) Barabacci, M.R., "A comparison of register transfer languages for describing computers and digital systems ", IEEE, Trans. on Computers, Vol. C-24. No.2, Feb. 1975, PP 137-150.

(2) Bell, C.G., and A. Newell, "Computer Structures: Readings and Examples ", McGraw Hill Book Co.,1971.

(3) Bell, C.G., and A. Newell, "The PMS and ISP Descriptive Systems for Computer Structures", AFIPS, SJCC, 1970, P.351.

(4) Bell, C.G., and J. Grason, "The RTM design concept ", Computer Design, Vol. 10, PP 87-94, May 1971.

(5) Breuer, M.A., "A survey of design automation", Computer Journal, 1972.

(6) Breuer, M.A., "General Survey of the design automation of digital computers ", Proc. IEEE Vol. 54, PP. 1708-1721, 1966.

(7) Chu, Y., "An ALGOL like computer design language", CACM, Vol. 8, Oct. 1965, PP. 607-615.

(8) Chu, Y., "Introduction to Computer Organization.", Prentice Hall Inc., Englewood Cliffs., N.J., 1970.

(9) Chu, Y., "Computer Organization and Microprogramming ", Prentice Hall Inc., Englewood Cliffs., N.J., 1972.

(10) Crockett, E.D., and et. al., "Computer aided system design ", AFIPS, FJCC, 1970, P-287.

(11) Duley, J.R., and D.L. Dietmeyer, "A digital system design language (DDL)" , IEEE Trans. on Computer, Vol. C. 17, Sept. 1968, PP 850-861.

(12) Duley, J.R., and D.L. Dietmeyer, "Translation of a DDL digital system specification to Boolean equations ", IEEE Trans. on Computer, April, 1969, PP. 305-318.

(13) Eckhouse, Jr., R.H., "A high level microprogramming language (MPL) ", AFIPS, SJCC 1971, P. 169.

(14) Falkoff, A.P., Iverson, K.E., and Sussenguth, E.H., "A formal description of system/360 ", IBM Systems J., Vol. 3, PP 198-262, 1964.

(15) Gerald, G.B., "Digital system design automation - A method for designing a digital system as a sequential network system.", IEEE Trans. on Computer, 1968, PP: 1044-1061.

(16) Gries, D., "Compiler Construction for Digital Computers". John Wiley and Sons Inc. 1971.

(17) Hill, F.J., and G.R. Peterson, "Digital Systems : Hardware Organization and Design ", Wiley, New York, 1973.

(18) "IBM-1800 Functional Characteristics "- IBM Systems Reference Library, File No. 1800-01, Form A 26-5918-7.

(19) "IBM 7040-7044 Principles of Operation"- IBM Systems Reference Library, File No. 7040/7044-01, Form A22-6649-6.

(20) Iverson, K.E., "A Programming Language ", Wiley, New York, 1962.

(21) Isloor, S.S., "A Syntax Directed Interpreter For Digital System Design Language ", M. Tech. Thesis, Computer Science Program, I.I.T. Kanpur, July, 1975.

(22) Kanodia,R.K., "Software for a small on line computer", M.Tech. Thesis, Electrical Engg., I.I.T. Kanpur, August,1975.

(23) Krishnamurthy, S.G., "Digital System Simulation and Design Language (DSDL) ", M. Tech. Thesis, Electrical Engg., I.I.T. Kanpur, August, 1973.

(24) Kutzberg, J.M., and et. al., "Computer design automation: What now and what next ? " AFIPS, FJCC, Vol. 33, Part 2, 1968, PP : 1499-1503.

(25) Metze, G., and Seshu, S., "A proposal for computer compiler ", AFIPS, SJCC, 1966, P. 253.

(26) Nori, K.V., "Introduction to compiler construction", NCSDCT, TIFR, BOMBAY, August 1975.

(27) Proctor, R.M., "A logic design translator experiment demonstration relationships of language to systems and logic design " IEEE Trans. on Computer, Vol. EC-13, August 64, PP-422-430.

(28) Rajaraman, V., and T. Radhakrishnan, "An introduction to digital computer design ", Prentice Hall of India, New Delhi, 1975.

(29) Schalappi, H.P., "A formal language for describing machine logic, timing and sequencing (LOTIS) ", IEEE Trans. on Computer, P. 439, 1964.

(30) Shinghal, R., "Simulation of TDC-12 on IBM. 7044", M.Tech. Thesis, Electrical Engg., I.I.T. Kanpur, June 1968.

(31) Stabler, E.P., "System Description Languages", IEEE Trans. on Computer, Dec. 1970, PP : 1160-1173.

(32) "System TDC-316- Trading Manual " Computer Group, ECIL, Hyderabad, Sept., 1974.

(33) Wirth, N., and K. Jensen, "Manual for learning the language PASCAL " Zurich, April 1974.

(34) Zucker, M.S., "LOCS : An EDP machine logic and control simulator ". IEEE Trans. on Computer 1965, P : 304.

- * -

APPENDIX I

TDC-316 CSDL DESCRIPTION

```
{ Braces are used for comments in the following description}
  SYSTEM  TDC-316                              { Heading}

 { MEMORY}
   MEMORY        M(0:32767; 15:0)

           { 32K words core memory, of which first 28K is user
             accessible. Remaining 4K words are reserved for
             processor registers, I/O registers and Rom modules.
             Here MSB is bit '15' and LSB is bit '0'}

 SUBMEMORY     MD1(0:8191;15:0) = M(0:8191;15:0)
               MD2(0:8191;15:0) = M(8192:16383;15:0)
               MD3(0:8191;15:0) = M(16384:24575;15:0)
               MD4(0:4095;15:0) = M(24576:32767 ;15:0)

 { User accessible core (28K) is further divided in 4 modules of
   8,8,8 and 4K each.}

{PROCESSOR }

           {There are 15 GPRs of which 3 are reserved for Program
            counter, User stack Pointer and Supervisor stack pointer.
            Remaining GPRs are divided as set '0' and Set '1' GPRs }

ARRAY      GPR(1:7), GPR1(1:7)

EQUIVALENT  PC = M(32752),                   {Program Counter }
            GPR(1) = M(32753),              {Supervisor Stack Pointer}
            GPR1(1)= M(32761),               {User Stack Pointer}
              {Set '0'}                            {Set '1' }
            GPR(2) = M(32754),    GPR1(2) = M(32762),
            GPR(3) = M(32755),    GPR1(2) = M(32763),
            GPR(4) = M(32756),    GPR1(3) = M(32764),
            GPR(5) = M(32757),    GPR1(4) = M(32765),
            GPR(6) = M(32758),    GPR1(5) = M(32766),
            GPR(7) = M(32759),    GPR1(6) = M(32767),

           { Above two sets can be used as an accumulator, index
            register, stack pointers, and locations for temporary
            storages }
```

```
                   PS = M(32736),  { Processor status register }
                   SL = M(32738),  { Stack limit register }
                   MQ = M(32739),  { Multiplier-quotient register }
                   SR = M(32737),  { Switch register }
                   SC = M(32751)   { Shift counter }

REGISTER           STOP,             PARITY-ENABLE-FLAG,
                   C,S,Ø, Z         {Condition Codes}

INTEGER            ZZ, Z1, Z2, Z3, PCT, D, S

{CONSOLE }

REGISTER           CPUST (17:0),{ 15 lamps to indicate various CPU states}
                   PWR,          {Power ON/OFF switch}
                   BUSADDR (17:0) {Bus-address register display}
                   DATAREG(17:0), {Data Register Display }
                   SWREG (17:0),  {Switch register }
                   CADRSEL(4:0),  {Console address select knob}
                   CONTRSW (10:0) {Control Switches}

{ INSTRUCTION FORMAT }

REGISTER        IR(15:0)                   { Instruction Register }

SUBREGISTER     DST(5:0) = IR(5:0),        { destination field}
                SRC(5:0) = IR(11:6),       { source field }
                REGD (2:0)= IR(2:0),       { destination register }
                MODED(2:0)= IR(5:3),       { destination mode }
                REGS(2:0) = IR(8:6),       { Source Register }
                Modes(2:0) =IR(11:9)       { Source mode }

{ Double Operand Class}

SUBREGISTER  OPD(3:0) = IR(15:12)  { OP-code }

{Single Operand Class}

SUBREGISTER   OPS(9:0) = IR(15:6)   {OP code}

{Branch Class}

SUBREGISTER    SINDIS (7:0) = IR(7:0), {signed displacement}
               OPB(7:0) = IR (15:8)    {OP-Code}

 {Condition Code Setting and Miscellaneous Operational Class }

SUBREGISTER    OPO(15:0) =IR(15:0)     {OP-Code}
```

```
{ Subroutine Linkage Class }

  SUBREGISTER        LINKR(2:0) = IR(8:6), { Linkage Register }
                     OPSBL(6:0) = IR(15:9) { Subroutine Linkage op-code }

{ Trap Class }
  SUBREGISTER        TRPRG(7:0) = IR(7:0) , { Trap Argument }
                     OPTR (7:0) = IR(15:8)     {Op-code }
```

<u>DATA FORMAT</u>

Z is integer variable

```
  SUBREGISTER     DATA (14:0) = M(Z, 14:0),     { data Part }
                  SW          = M(Z, 15),       {Sign bit of the word }
                  HIGHBYT(15:8)=M(Z,15:8),       { Higher byte }
                  SHB          = M(Z, 15),         { Sign bit }
                  DATAH (14:8) =M(Z,14:8),       { data part }
                  LOWBYT(7:0) = M(Z,7:0),       { Lower Byte }
                  SLB          = M(Z,7),          { Sign bit }
                  DATAL (6:0) = M(Z,6:0)        { data Part }
```

{<u>ADDRESS FORMAT</u>}

```
REGISTER          ADDRWRD (15:0)                 { address word }

SUBREGISTER       B = ADDRWRD(0),               {Byte-manipulation bit }
                  ADDR(15:1) = ADDRWRD(15:1)    { Word address }
```

{LSB of Address word (ADDRWRD) is a bit 'B', which is used in byte manipulation instructions only. If B = 0 then lower byte (7:0) of the word is selected and if B=1 then higher byte(15:8) of the word is selected }

{ <u>EFFECTIVE ADDRESS CALCULATION</u> }

```
PROCEDURE     EFFADDR (MODE,REG, ZZ)

BEGIN
  IF(PC-MODE)  THEN   PCMADDR          { Processor-mode of addressing }
               ELSE GPRMADDR           { GPR- mode of addressing }

END
```

```
PROCEDURE GPRMADDR                    { GPR-mode of addressing }

BEGIN

{ Direct Addressing }

 CASE MODE OF

000:  BEGIN                           { Register Operand }

              CASE REG OF

              1: BEGIN  ZZ = 32753
              2: BEGIN  ZZ = 32754
              3: BEGIN  ZZ = 32755
              4: BEGIN  ZZ = 32756
              5: BEGIN  ZZ = 32757
              6: BEGIN  ZZ = 32758
              7: BEGIN  ZZ = 32759

              CASEND

       END

001·  BEGIN                           { Register Increment}

              ZZ = GPR(REG)
              GPR(REG) = GPR(REG)+ 2

      END

002:  BEGIN                           { Register decrement}

              ZZ = GPR(REG)
              GPR(REG) = GPR(REG) - 2

      END

{ Indexed Addressing }

003:  BEGIN

              ZZ = GPR (REG)+ M(PC+2)

      END
```

```
{ Deferod Addressing }

004:  BEGIN

            ZZ = GPR(REG)                   { Register addressing }

      END

005:  BEGIN                                 { defered register increment }

               Z1 = GPR(REG)
               ZZ = M(Z1)
               GPR(REG) = GPR(REG) + 2

      END

006:  BEGIN                                 { defered register decrement}

               CPR(REG) = GPR(REG)-2
               Z1 = GPR(REG)
               ZZ = M(Z1)

      END

007:  BEGIN                                 { defered indexed }

               Z1 = M(PC +2)
               Z2 = GPR(REG)
               Z3 = Z1 + Z2
               ZZ = M(Z3)

      END

 CASEND
 ADDRWRD = ZZ
 ZZ = ZZ/2

 END

 PROCEDURE      PCMADDR                     { Processor mode of addressing}

 BEGIN

     CASE MODE OF

 001 :  BEGIN                               {Immediate}
              ZZ = PC
              PC = PC+2

        END
```

```
003:  BEGIN                                 { Relative}
              ZZ = M(PC) + PC
              PC = PC + 2

      END

005:  BEGIN                                 { absolute}

              ZZ = M(PC)
              PC = PC+2

      END

007:  BEGIN                                 { defered relative}
             Z1 = M(PC) + PC
             ZZ = M(Z1)

      END
CASEND

ADDRWRD = ZZ

ZZ = ZZ/2

END
```

{ INSTRUCTION EXECUTION }

{ Following procedure executes the instruction }

```
 PROCEDURE INSTRXEQ
```

{ This procedure calls different procedures depending upon bits of IR }

```
 IF(IR(15:03) = 0)  THEN OPRCL          {Operational Class}

                    ELSE

 IF(IR(15:03) =1)   THEN RTRSUB         { Return from Subroutine}

                    ELSE

 IF(IR(15: 6) = 0)  THEN  ADDCL         {additional class}

                    ELSE
```

```
IF(IR(15:6) =1)   THEN  CONDSET        { Condition code setting }
                  ELSE
IF(IR(15:7) =3)   THEN  TRAP           { Trap }
                  ELSE
IF(IR(15:8) =1)   THEN IOTRAP          { I/O  Trap}
                  ELSE
IF(IR(15:8) =2)   THEN EMUTR           { Emulator Trap}
                  ELSE
IF(IR(15:9) =3)   THEN SUBLNK          { Subroutine Linkage }
                  ELSE
IF(IR(15:12)=1)   THEN BRANCH          { Branch Class }
                  ELSE
IF(IR(15:12)=0)   THEN SINGOPR         { Single operand class }
                  ELSE     D6UBOPR     { Double operand class }

END

PROCEDURE         DOUBOPR    { Double Operand Instruction execution}

REGISTER BD, BS          {Byte manipulation bits for destination
                              source operands}

CONCAT            BDBS = BD°BS

BEGIN

    EFFADDR (MODED, REGD, D)       {destination effective address}

    BD = B
    EFFADDR (MODES, REGS,S)        {Source effective address}

    BS = B
```

```
CASE OPD  OF                    { OPD  is double  operand op-code }

02 : BEGIN                           { Set  Bit : STB }

        M(D) = M(D) V M(S)

     END

03 : BEGIN                      {Set   byte : BSTB}

          CASE  BDBS  OF

          00 :  BEGIN  M(D; 7:0) = M(D; 7:0) V M(OS;7:0) END

          01 :  BEGIN  M(D; 7:0) = M(D; 7:0) V M(S; 15:8)END
          02 :  BEGIN  M(D;15:8) = M(D;15:8) V M(S;7:0)  END
          03 :  BEGIN  M(D;15:8) = M(D;15:8) V M(S;15:8) END

END
  { Above byte manipulation instruction is illustration. Hereafter
    byte manipulation instructions are not explained in details.}

04 :  BEGIN                          { Compare : CMP }

          M(D) = M(S)- M(D)

      BEGIN

      IF(M(D) < 0)  THEN  S=1  ELSE   S=0
      IF(M(D) =0)   THEN  Z=1  ELSE   Z=0
      IF(CARRY=1)   THEN  C=0  ELSE   C=1
      IF(OVFL= 1)   THEN  Ø=1  ELSE   Ø=0

      END

     END

 05: {Byte Compare : BCMP}     {Same as CMP  for Byte}

 06:  BEGIN                    {Test  Bit : TSB}

         M(D) = M(S)ΛM(D)
         BEGIN
         Ø = 0
         IF(M(D) = 0) THEN Z=1 ELSE Z = 0
         IF(M(D;15)=1) THEN S=1 ELSE S= 0
         END

      END
```

```
07 : {Byte Test Bit : BTSB }  {Same as TSB for Byte}

10 :  BEGIN                   {Multiply : MPY}
      M(D)°MQ = M(S)*MQ
      END

11 :  BEGIN                   {Divide : DVD}
      MQ°M(D) = (M(D)°MQ)/M(S)
      END

12 :  BEGIN                   {Transfer : TSR }
      M(D) = M(S)
      END

13 : {Byte Transfer : BTSR }

14 :  BEGIN                   { add: ADD }
      M(D) = M(D) + M(S)
      END

15 :  BEGIN                   { Subtract : SUB }
      M(D) = M(D) - M(S)
      END

16 :  BEGIN                   { Clear Bit : CLB }
      M(D) = M(S)/\M(D)
      END

17 :{ Clear byte : BCLB }

CASEND
END

PROCEDURE  SINGOPR               { Single Operand class instruction
                                          execution process

BEGIN
      EFFADDR (MODES,REGS, D)
```

```
IF(IR(11:10) = 1) THEN SINGOPRRS           {Single Operand Rotate Shift}
                  ELSE SINGOPRGL           {Single Operand General}
END
PROCEDURE  SINGOPRGL
   CASE  OPS  OF                 {OPS(9:0) = IR(15:6) : OP-code}
60:  BEGIN M(D) = 0 END          {Clear : CLR}
61:  {Byte clear : BCLR}
62:  BEGIN                       {test : TST}
       BEGIN
     IF(M(D) = 0) THEN Z = 1 ELSE Z = 0
     IF(M(D) < 0) THEN S = 1 ELSE S = 0
     C = 0
     O = 0
     END
   END
63 : {Byte Test : BTST }
64 : BEGIN                        { Increment : INC }
            M(D) = M(D) + 1
     END
65 :{ Byte increment : BINC }
66 : BEGIN                        { Add Carry : ADC }
     M(D) = M(D) + C
     END
67 :{ Byte Add Carry : BADC }
70 : BEGIN                        { Decrement : DEC }
     M(D) = M(D)-1
     END
71 :{ Byte Decrement : BDEC }
72 : BEGIN                        { Subtract Carry : SBC }
     M(D) = M(D) - C
     END
73 : {Byte Subtract Carry : BSBC}
```

```
74  : BEGIN                           { negate : NEG}
      M(D) =  M(D) + 1
      END
75  : { Byte Negate : BNEG }
76  : BEGIN                          { Complement : COM }
      M(D) = ¬M(D)
      END
77  : { Byte Complement : BCOM }
CASEND
PROCEDURE  SINGOPRSR                 { Single Operand Shift Rotate }
 CASE  OPS  OF                      { OPS = Single Operand Group Op-code }
40:: BEGIN                          {Rotate Right : ROR }
      ↻ CoM(D)
      END
41  :{ Byte Rotate Right : BROR }
44  : BEGIN                          { Rotate Left : ROL}
      ↺ CoM(D)
      END
45: { Byte Rotate Left : BROL}
46  : BEGIN                           { Long Rotate Left : LROL}
      ↺ C∘ M(D)oMQ
      END
47  : BEGIN                           { Long Rotate Right : LROR}
      ↻ C o M(D)oMQ
      END
50. : BEGIN                            {Arithmetic Shift Right.ASR }
      ↳ $M(D)_oC$
      M(D;IS) = M(D;14)
      END
51  : { Byte arithmetic Shift Right : BASR }
52  : BEGIN                          { Long Arithmetic Shift Right : LASR }
      FOR I = 1 TO SC DO
      BEGIN
      ⊥→ M(D)°MQ°C
          M(D;15) = M(D;14)
       END
     END
54 : BEGIN                       {Logical Shift Left : LSL}
      ←⊥ C°M(D)
      M(D;0) = 0
      END
55 : {Byte Logical Shift Left : BLSL }
56 :  BEGIN                      {Long Logical Shift Left : LLSL }
       FOR I = 1 TO SC DO
      BEGIN
      ←⊥ C°M(D)°MQ
       MQ(0) = 0
       END
      END
57 :  BEGIN                      { Long Normalize : LNMI }
       SC = 0
       REPEAT
             BEGIN
             ←⊥  M(D)°MQ
               SC = SC + 1
               END
      UNRIL (¬ M(D;15) = M(D;14)
     END
03 :  BEGIN                      {Interchange byte:INIB }
      FOR I = 1 TO 8 DO
       BEGIN
        ←⊥  M(D)
       END
     END
02 : BEGIN                        {Jump : JMP}
      PC = M(D)
```

```
        END
    CASEND
    PROCEDURE  BRANCH               {Branch Class }
    {First 8 bits are op-code & remaining 8 bits represent}
    {Signed displacement  OPB(7:0) = IR(15:3),XX(7:0)=IR(7 0)}
    CASE   OPR   OF
100:  BEGIN                         {Branch on count non zero : BRCH }
      SC = SC - 1
      IF(SC≠0) THEN  PC = PC + 2*XX
      END
104:  BEGIN                         { Branch on zero clear or Branch on not}
                                               equa: BRZC (BRNE) }
      IF(Z=0) THEN PC = PC+ 2*XX
      END
110:  BEGIN                         { Branch on greater or equal  : BRGE}
      IF(C↑V= 1) THEN  PC = PC + 2*XX
      END
114:  BEGIN                          {Branch on greater than  : BRGT }
      IF(Z/\(C⊕V) = 0)  THEN  PC = PC + 2*XX
      END
120:  BEGIN                          { Branch unconditional: BRN }
     PC = PC + 2*XX
      END
121:  BEGIN                           { Branch on Zero Set  : BRZS }
      IF(Z=1)  THEN  PC = PC  + 2*XX
      END
130:  BEGIN                            { Branch on less than :BRLT }
     IF(S⊕V= 1)  THEN  PC = PC + 2*XX
      END
131:  BEGIN                            {Branch on less than or equal 10:BRLE }
      IF((S⊕V)/\Z=1)  THEN  PC = PC+2*XX
      END
```

```
140 :   BEGIN                   { Branch on plus or sign clear : BRSC(BRPL) }
        IF(S=0) THEN  PC=PC+2*XX
        END

141 :   BEGIN                   { Branch on overflow clear : BROC }
        IF(Ø= 0)  THEN  PC = PC + 2*XX
        END

150 :   BEGIN                   { Branch on Carry Clear : BRCC }
                                { or Branch on higher or same : BRHS }
        IF (C=0) THEN  PC = PC + 2*XX
        END

151 :   BEGIN                   { Branch on Higher : BRHI }
        IF(C↑Z = 1 )  THEN  PC = PC + 2*XX
        END

160 :   BEGIN                   {Branch on Sign Set or minus}
                                  { BRSS ( BRMI )}
        If(S=1) THEN PC = PC + 2*XX
        END

161 :   BEGIN                   {Branch on overflow Set : BROS}
        IF (Ø= 1 )  THEN  PC = PC + 2*XX
        END

170 :   BEGIN                   {Branch on Carry Set : BRCS
                                  "       " Lower    : BRLO}
        IF ( C = 1 ) THEN  PC = PC + 2*XX
        END

171 :   BEGIN                   {Branch on lower or Same : BRLS}
        IF ( CVZ = 0 )  THEN  PC = PC + 2*XX
        END
CASEND
PROCEDURE  SUBLNK                    { Subroutine Linkage }
{ OPSBL = IR (15:9) is already tested }
SUBREGISTER   RLINK (2:0) = IR(8:6)
BEGIN
  EFFADDR ( MODES,REGD,D )
       STACK = GPR(RLINK)
        GPR(RLINK) = PC
        PC  =  M(D)
END
PROCEDURE  EMUTR                    { Emulator Trap }
BEGIN
     STACK = PS                     { Argument TT = IR(7:0) provides
     STACK = PC                       necessary information for
     PC  =  M(16)                     Trap SIR }
     PS  =  M(18)
END
PROCEDURE   IOTRAP                  {I-O Trap}
BEGIN
     STACK = PS
     STACK = PC
     PC  =  M(20)
     PS  =  M(22)
END
```

```
PROCEDURE   TRAP                         { TRAP }
BEGIN
      STACK = PS
      STACK = PC
      PC = M(8)
      PC = M(10)
END

PROCEDURE   CONDSET                  { Condition Code Setting }
 { OPO  =  IR(15:0)  :  OP code }
BEGIN
  CASE  OPO  OF
101  : BEGIN C = 0 END        { Clear Carry Bit      :   CLC }
102  :  "     Ø = 0  "        {  "   Overflow Bit   :   CLO }
104  :  "     Z = 0  "        {  "   Zero Bit       :   CLZ }
110  :  "     S = 0  "        {  "   Sign  Bit      :   CLS }
120  :  "     B = 0  "        {  "   Break Point Bit:   CBPB}
141  :  "     C = 1  "        {  " Set Carry Bit    :   STC }
142  :  "     Ø = 1  "        {  " Set Overflow Bit:    STO }
144  :  "     Z = 1  "        {  " Zero Bit         :   STZ }
150  :  "     S = 1  "        {  " Sign Bit         :   STS }
160  :  "     B = 1  "        {  " Breakpt. Bit     :   STBP}
137  : BEGIN                     { Clear Conditional Codes : CLCC }
         BEGIN
           C = 0
           Z = 0
```

```
        ∅  = 0
        S  = 0
        B  = 0
      END
    END
100:    BEGIN                        { No Operation : NOP }
        END
140:    BEGIN    END                 { No Operation : NOP }
CASEND
END
PROCEDURE  RTRSUB                  {{ Subroutine Return Class }
SUBREGISTER  RDST ( 2:0) = IR(2:0)
BEGIN
   PC = GPR (RDST)
   GPR(RDST) = M(SP)
    SP  =  SP + 2
END
PROCEDURE   OPRCL                  { Operational Class}
 OPO(15:0) = IR(15:0)
CASE   OPO   OF
000: BEGIN   STOP = 1  END            { Processor Halts
                                                : STOP }

001: BEGIN   WAIT = 1 END              { Wait
                                           :WAIT }

002: BEGIN                           { RTI  : Exit from intempt
         PC = STACK                     routine or Trap
         PS = STACK                     Service routine }
     END
003: BEGIN                            { Break point Trap : BPT }
       STACK = PS
       STACK = PC
       PC = M(12)
       PS = M(16)
 END

006:BEGIN                          {PDB : Parity disable}

    PARITY-ENABLE-FLAG = 0

 END

007:BEGIN                          { PEN : Parity enable}

    PARITY-ENABLE-FLAG = 1

 END

CASEND
END
```

{ <u>INSTRUCTION INTERPRETATION</u>}

```
    BEGIN

      REPEAT
    BEGIN
         PCT = PC/2
         IR = M(PCT)
         INSTREQ
          PC = PC+2
     END
       UNTIL
          STOP = 0
     END
```

– * –

APPENDIX II

IBM - 1800 CSDL DESCRIPTION

```
{ IBM-1800  description in CSDL }

 SYSTEM     IBM-1800                          { Heading }

{ MEMORY }

 MEMORY     M(0:16383 ; 0:15, P,S)
  { Memory consists of  16K words of 16 bits each.  Each word is
    associated with parity (P) and Memory protect (s)  bit, thus
    moking 18 bits words }

{ PROCESSOR }

 REGISTER          SAR(0:15),          { Storage address register }
                   I(0: 15),           { Instruction Register }
                   B(0: 15),           { Storage Buffer Register }
                   D(0: 15),           { Arithmetic Factor Register }
                   A(0: 15),           { Accumulator }
                   Q(0: 15),           { Accumulator Extension }
                   SC(0: 5) ,          { Shift Control Counter }
                   OP(0: 5) ,          { Op-code register holds op-code }
                   OV, C,              { Overflow and Carry indicators }
                   XR (1:3;0:15)       { 3 Index Registers 1,2 and 3 }

 INTEGER           Z,Z1, ZZ             {Integer variables}

{ CONSOLE }

 REGISTER          CLEARSTORE, PROG-LOAD,   {Name of switch indicates}
                   READY, ON, OFF, POWER,    its use also}
                   LAMP-TEST, WAIT, RUN,
                   ALARM, EMER-PULL, CONSOLE
                   INTERRUPT, LOAD, RESET,
                   IMMEDSTOP, START, STOP

{INSTRUCTION FORMAT}

 REGISTER          CR(0:1; 0:15)        {Control register }

 SUBREGISTER       OP(0:4) = CR(0;0:4),  {Op-code}
                   SHOP(0:7)=CR(0;0:4,5,8,9), {Shift op-code}
                   F        =CR(0;5),    {Format: Short or Long }
                   T(0:1)   =CR(0:6,7),  {Tag Bits : indexing }
```

```
DISP(0:7) = CR(0; 8:15),            { displacement }
ADDR(0:15)= CR(1; 0:15),       { long instruction address}
IA        = CR(0; 8) ,            { Indirect addressing }
BO        = CR(0; 9) ,                 { Branch out}
COND (0:5)= CR(0;10:15)                 { Condition}
```

{ <u>IOCC FORMAT</u> }

{ Input Output Control Code instruction format is explained below}

```
REGISTER   IOCC(0:15)

SUBRFGISTER      AREA (0:4) = IOCC (0:4),
                 FUN  (0:2) = IOCC (5:7),
                 MODIF(0:7) = IOCC (8:15)
```

{ Here AREA gives a unique segment of I/O which may be single device or a group of several devices. FUN gives primary I/O function. MODIF gives the extension OF I/O function code or AREA code}

```
PROCEDURE EFFADDR (Z)
             {This procedure calculates the effective address when
BEGIN         the instruction in instruction register is being decoded}
      IF(T=0) THEN               { T = Tag}
              BEGIN
      IF (F=0) THEN Z = DISP + I                { Direct Addressing }
              ELSE
      IF (IA=0)THEN Z = ADDR                    { Direct Addressing }
                ELSE Z = M(ADDR)                { Indirect Addressing}

 EISE            END

 BEGIN
 IF(F=0) THEN Z = DISP + XR(I)                  {{Direct}
         ELSE
 IF (IA=0) THEN Z = ADDR + XR(T)                 {Direct Addressing}
    ELSE BEGIN  Z1= M(ADDR + XR(T)               {Indirect }
                Z = M(Z1)
         END

 END
END
```

```
PROCEDURE          INSTRXEQ              {Instruction Execution}
SUBREGISTER        TEST(0:3) = IR(0:3)
BEGIN
IF (TEST= 0)       THEN EXECIOCC         {IOCC 2 execute}
IF (TEST= 1)       THEN SHIFTCL          {Shift cbss}
{ Following part executes the instructions left }
EFFADDR(Z)                               { Effective address calculation }
 ZZ = Z + 1
CASE   OP   OF                           { OP = Opcode = IR(0:4) }
{ Arithmetic Instructions }

030:  BEGIN                              { Load accumulator :LD }
      A = M(Z)
      END
31 :  BEGIN                              { Load Acc. and Q : LDD }
      A°Q = M(Z)°M(ZZ)
      END

32 :  BEGIN                              { Store accumulator: STO }
      M(Z) = A
      END

33 :  BEGIN                              { Store A and Q : STD }
      M(Z)°M(ZZ) = A°Q
      END
20 :  BEGIN                              { Add : A }
      OV°C°A = A + M(Z)
      END

23 :  BEGIN                              { Subtract double : SD }
      A°Q = A°Q - M(Z)°M(ZZ)
      END

22 :  BEGIN                              { Add double : AD }
      A°Q = A°Q + M(Z)°M(ZZ)
      END

24 :  BEGIN                              { Multiply: M}
      A°Q = A*M(Z)
      END

25 :  BEGIN                              { Divide : D}
      Q = A°Q/M(Z)
      END
```

```
{ Logical Instructions }

 34 :  BEGIN
       A = A  M(Z)                        { AND }
       END

 35    BEGIN
       A = A∨ M(Z)                        { OR }
       END
 36 :  BEGIN                              { Exclusive OR : EOR }
       A = A + M(Z)
       END


{ Compare Instructions }

 26 :  BEGIN                              { Compare : CMP }
       IF (A <M(Z)   THEN I=I+1
       IF (A =M(Z)   THEN I=I+2
       END

 27 :  BEGIN                              { Double Compare : DCM }

       IF((A°Q) <(M(Z)°M(ZZ)) THEN I = I+1
       IF((A°Q) =(M(Z)°M(ZZ))) THEN I = I+2
       END

 14 :  BEGIN                              { Load index or instruction counter:LDX }

       IF(T=0) THEN I = ZZ
               ELSE XR(T) = ZZ

       END

 15 :  BEGIN                    { Store, index or instruction counter: STX }

       IF(T=0) THEN M(ZZ) = I
               ELSE M(ZZ) = XR(T)

       END

 05 :  BEGIN                              {Store Status : STS}
       IF (F∧BO =1)   THEN      M(Z; P) = COND(15)
       IF (BO=0)    THEN BEGIN  M(Z; 14:15) = C°OV
                                C°OV = 0
                                END
       END
```

```
{Shift Instructions}

PROCEDURE SHIFTCL
BEGIN
IF (T=0) THEN  SC(0:5) = CR(0;10:15)
        ELSE  SC(0:5) = XR(T;10:15)

CASE SHOP OF

020 : BEGIN                              {Shift left logical : SLA}
      FOR  Z1 = 1 TO SC DO
      BEGIN
      ↲  A
      END
      C = A(SC-1)
      END

0.22: BEGIN                              {Shift double left logical: SLT}
      FOR Z1 = 1 TO SC DO
      BEGIN
      ↲  A°Q
      END
      C= A(SC-1)
      END

030 : BEGIN                              {Shift right logical : SRA}
      FOR Z1 = 1 TO SC DO
      BEGIN
      ↳  A
      END
      END

032 : BEGIN                              {Shift right A & Q : SRT}
      FOR Z1 = 1 TO SC DO
      BEGIN
      ↳ A°Q
      END
      END

033 : BEGIN                              {rotate right A & Q : RTE}
      FOR Z1 = 1 TO SC DO
      BEGIN
            A(0) = Q(15)
        ↳ A°Q
      END
      END
```

```
021 :  BEGIN                           {Shift left and count A: SLCA}
       IF (T = 0) THEN BEGIN
                       FOR Z1 = 1 TO SC DO
                       BEGIN
                        ← A
                       C = A(SC-1)
                       END
                       END
            ELSE  BEGIN
                  REPEAT BEGIN
                        ← A
                        SC=SC-1
                        IF(SC=0)THEN STOP=1
                        END
                  UNTIL ((¬ A(0))V(¬STOP))
                  STOP= 0
                  XR(T;10:15)=SC(0:5)
                  XR(T;8:9)=0
                  END
```

```
PROCEDURE        BROINT             { Branch out of Intempt}
BEGIN
    IF (SKIPCOND∧(¬F) = 1)  THEN  BEGIN
                                  I = I+1
                                  INTERRUPT =1
                                  END
    IF( (¬ SKIPCOND ∧ F) = 1) THEN  BEGIN
                                  I=Z
                                  INTERRUPT=1
                                  END

    IF(D(15)= 1)  THEN  OV = 0

 END

PROCEDURE EXECIOCC
{This procedure executes IOCC instruction. Here effective address
 must be even}

BEGIN
EFFADDR(Z)
CASE FUN OF

01:  BEGIN                       { Memory write operation
                                       AREA gives I/O device number}
     WRITE (AREA), M(Z)
     END

02:  BEGIN                       { Memory read operation
                                   AREA gives I/O device number}
     READ(AREA), M(Z)
     END
CASEND
   END

 {Statement part of main program}

  BEGIN
      REPEAT
            BEGIN
                IF(RUN)THEN
                            CR(0:1) = M(I)°M(I+1)
                IF(F=1) THEN I= I+2
                        ELSE I=I+1
                END
          UNTIL  (STOP = 0)
END
```

- * -

APPENDIX III

IBM - 7044 CSDL DESCRIPTION

```
{ Following is the CSDL  description of  IBM 7044}

  SYSTEM        IBM-7044                 {Heading}

{ Memory }

  MEMORY        M(0:32767; S,1:35)
            {32K words core memory of 36 bits word each. 'S' is
                                                        sign bit}

{ Processor }

  REGISTER      AC(S,Q,P,1:35)       { 38 bits Accumulator}

  SUBREGISTER ACS(S,1:35) = AC(S,1:35),{ signed  AC word}
              ACL(P,1:35) = AC(P,1:35),{ logical AC word}
              P = AC(P);               { carry for AC (1:35)}
              Q = AC(Q);               { carry for AC(P,1:35)}
              S = AC(S),               { sign bit}
              ACQP(Q,P,1:35) = AC(Q,P,1:35)

  REGISTER      MQ(S,1:35)                 {Multiplier Quotient}

  CONCAT        ACMQ(S,Q,P,1:71) = AC°MQ(1:35)
                                           {double word accumulator}

  REGISTER      SI(0:35),           {sense indicators for floating point
                                                     instructions}

                XR(1,2,4;3:17),     {index registers 1,2 and 4 or
                                                     A,B and C }

                IC(3:17),           { instruction location counter }
                RUN,                { indication whether machine is
                                                   executing }

                DIVCK,              {divide check }
                ACOVFL,             { Accumulator overflow }
                MQOVFL,             { MQ overflow }
                IOCK,               { Input output check }
                AR(1:15)            { Address register }

  INTEGER       Z, ZD
```

```
{ Console }

  REGISTER         KEYS(0:35),      { console data }
                   SENSSW(0:5),     { sense switches}
                   SENSLT(0:3)      { sense lights }

{Instruction Format }

  REGISTER         IR(S,1:35)        {Instruction register }
                   Y(1:15)=IR(21:35), {address part: word address }
                   T(1:3) =IR(18:20), {Tag: XR to use 1,...7.}
                                      { '0' means no indexing}
                   F(1:2) = IR(12:13), {Indirect addressing bits}
                                        If F(1:2)=11 then indirect}
                   OP(S,1:11) = IR(S,1:11)   OP-code

 {Data Format}

  REGISTER         SR(S,1:35)        {storage register}

  SUBREGISTER      SL(S,1:35) = SR(S,1:35), {logical data: unsigned
                                             integer :boolean vector }
                   SX(S,1:35) = SR(5,1:35), {single precision fixed point
                   SXSGN = SR(S),            sign bit of SX }
                   SXMGND(1:35) = SR(1:35), {magnitude of SX}
                   SF(S,1:35) = SR(S,1:35), {single precision floating pt.}
                   SFSGN = SR(S),           {sign of SF }
                   SFEXP(1:8) = SR(1:8),    {exponent part}
                   SFMANT(0:26)= SR(9:35)   {mantisa part }

  EQUIVALENT       DF(0:1;S,1:35) =  M(Z)°M(Z+1)  {Double precision
                                                       floating pt.}

SUBREGISTER        DFSIGN = DF(0;S)          {sign bit }
                   DFEXP(1:8)= DF(0;1:8)     {exponent part }
CONCAT             DFMANT(0:537)= DF(0;9:35)°DF(1;9:35) { mantisa part }

{Following procedure calculates Effective Address}

PROCEDURE EFFADDR(Z)

BEGIN

    ADDR(ZD)       {This procedure calculates direct address}
    IF(F(1:2)=3)  THEN  BEGIN
                        IR = M(ZD)     {Indirect Addressing}
                        ADDR(ZD)
                        Z=ZD
                        END

                   ELSE
    Z = ZD                  {Direct addressing}
    END

PROCEDURE ADDR(ZD)

    {Here multiple tags are used to calculate effective address}

        CASE  T   OF
    0:  BEGIN  ZD = Y  END
    1:  BEGIN ZD = Y- XR(1)  END   {Index reg. 1 or A}
    2:  BEGIN ZD = Y-XR(2)   END   {Index reg. 2 or B}
    3:  BEGIN ZD = Y-(XR(1)V XR(2)){index reg. 1 &  2}
    4:  BEGIN ZD = Y-XR(4)   END   {Index reg. 4 or C}
    5:  BEGIN ZD = Y-(XR(1)VXR(4)) END {Index reg.1 & 4}
    6:  BEGIN ZD = Y-(XR(2)VXR(4)) END {Index reg.2 & 4}
    7:  BEGIN ZD = Y-(XR(1)VXR(2)VXR(3)) {Index reg.1 & 2 & 3}
       CASEND
END

PROCEDURE INSTRXEQ              {Instruction Execution}

EFFADDR(Z)                      {Effective address calculation}

CASE  OP  OF

{Arithmetic Instructions}

+0361 :  BEGIN                  {add and carry logical word: ACL}
         ACL = AC + M(Z)
         END

+0400 :  BEGIN                  {add algebraic : ADD}
         AC = AC+M(Z)
         END

+0767 :  BEGIN                  {accumulator left shift: ALS}
         FOR I=1 TO SC DO
         BEGIN  ACQP  END
         END

+0771 :  BEGIN                  {accumulator right shift: ARS}
         FOR I=1 TO SC DO
         BEGIN
          ⊥→ ACQP
         END
```

```
-500 :  BEGIN          { clear and add logical : CAL }
        AC=0
        ACL=ACL+M(Z)
        END

+500 :  BEGIN          { clear and add Acc: CLA }
        AC=0
        ACS=AC+M(Z)
        END

+502 :  BEGIN          { clear and subtract : CLS }
        AC=0
        AC=AC-M(Z)
        END

+0221:  BEGIN     -    { divide or proceed: DVP }
        ACMQ=ACMQ/M(Z)
        END

+0420:  BEGIN          { Halt and Proceed: HPR }
        RUN=0
        END

+0560:  BEGIN          { load MQ : LDQ }
        MQ=M(Z)
        END

-0763:  BEGIN          { logical left shift: LGL }
        FOR I=1 TO SC DO
        BEGIN
        ← ACMQ°MQ
        END


-0765 : BEGIN          { logical right shift: LGR }
        FOR I=1 TO SC DO
        BEGIN
        → ACQP°MQ
        END
       END

+0763 : BEGIN          { long left shift : LLS }
        FOR I=1 TO SC DO
        BEGIN
        ← ACQP°MQ(1:35)
        END
```

```
+0765 : BEGIN                      { long right shift : LRS}
        FOR I=1 TO SC DO
        BEGIN
         ⊥⇁  ACQP°MQ(1:35)
        END


+0200 : BEGIN                      { Multiply : MPY}
        ACMQ=MQ*M(Z)
        AC(Q,P)=0
        END

-0773 : BEGIN                      { Rotate MQ left :RQL}
        FOR I=1 TO SC DO
        BEGIN
         ↼⊥  MQ
        END


+0602 : BEGIN                      {store logical word: SLW}
        M(Z) = ACL
        END

+0621 : BEGIN                      {store address: STA }
        M(Z)=AC(21:35)
        END

+0622 : BEGIN                      {store displacement: STD}
        M(Z) = AC(3:17)
        END

-0625 : BEGIN                      { store instruction location counter:STL}
        M(Z)=IC
        END

+0601 : BEGIN                      { store: STO }
        M(Z)=ACS
        END

-0600 : BEGIN                      { store MQ  :STQ }
        M(Z)=MQ
        END

-1000 : BEGIN                      { store location and trap : STR }
        M(0)=0
        M(0)=IC
        MC=M(Z)
        END
```

```
-0000 :  Similar to STR
+0000 :     "     "  "

+0600 :  BEGIN                      { Store zero :  STZ}
         M(Z)=0
         END

+0402 :  BEGIN                      { Subtract:  SUB}
         AC= AC-M(Z)
         END

-0120 :  BEGIN                      {Transfer on minus : TMI}
         IF(AC(S)=1)  THEN  IC=Z
         END

-0100 :  BEGIN                      {Transfer on no zero: TNZ }
         IF(A ≠0)  THEN IC=Z
         END

+0120 :  BEGIN                      {Transfer on plus : TPL}
         IF(AC(S)=0) THEN IC=Z
         END

+0140 :  BEGIN                      {Transfer on overflow : TOV}
         IF(ACOVF)  THEN  BEGIN
                          ACOVF=0
                          IC=Z
                          END
         END

+0020 :  BEGIN                      {Transfer: TRA}
         IC=Z
         END

-1627 :  BEGIN                 {Transfer and store instruction location
                                                          counter: TSL}
         M(Z)=IC
         IC=Z+1
         END

+0100 :  BEGIN                      { Transfer on zero : TZE }
         IF(AC=0) THEN IC=Z
         END

0522  :  BEGIN                      { Execute : XEC }
         INSRXEQ(Z)
         END
```

{ Logical Operations }

```
-0320 :  BEGIN                     { AND to Accumulator : ANA}
         ACL=ACL∧M(Z)
         AC(S,Q)=0
         END

+0340 :  BEGIN                     {Compare Accumulator with storage:CAS}
         IF(AC<M(Z)) THEN IC=IC+2
         IF(AC =M(Z)) THEN IC=IC+1
         END

-0340 :  BEGIN                     {logical compare ACC. with storage: LAS}
         IF(ACQP<M(Z)) THEN IC=IC+2
         IF(ACQP= M(Z)) THEN IC=IC+1
         END

-0501 :  BEGIN                     { OR to Accumulator : ORA }
         ACL=ACL V M(Z)
         END
```

Extended Performance Set

```
+0774 :  BEGIN                     { Address to index true : AXT }
         XR(T)=Y
         END

+0535 :  BEGIN                     { Load complement of address in index: LAC }
         XR(T)=¬M(Y:21:35)+1
         END

-0535 :  BEGIN                   { Load Complement of Decrement in index: LDC }
         XR(T)=¬M(Y;3:17)+1
         END

+0534 :  BEGIN                     { Load index from address: LXA }
         XR(T)=M(Y;21:35)
         END

-0534 :  BEGIN                     { Load index from Decrement: LXD }
         XR(T)=M(Y;3:17)
         END

+0737 :  BEGIN                     { Place complement of address in index:PAC}
         XR(T)=¬AC(21:35)+1
         END
```

```
+0734 :  BEGIN                            {Place address in index : PAX}
         XR(T) = AC(21:35)
         END

-0737 :  BEGIN                            {Place Complement of Decrement
         XR(T) =  AC(3:17) + 1                       in Index: PDC}
         END

-0734 :  BEGIN                            {Place Decrement in Index: PDX}
         XR(T) = AC(3:17)
         END

+0754 :  BEGIN                            {Place Index in Address: PXA}
         AC=0
         AC(21:35) = XR(T)
         END

-0754 :  BEGIN                            {Place Index in Decrement  : PXD}
         AC=0
         AC(3:17) = XR(T)
         END

+0634 :  BEGIN                            {Store Index in Address: SXA}
         M(Y; 21:35)  : XR(T)
         END

-0634 :  BEGIN                            {Store Index in Decrement: SXD}
         M(Y; 3:17) = XR(T)
         END

+0074 :  BEGIN                            {Transfer and Set Index : TSX}
         XR(T) = IC+1
         IC=Y
         END

 {Character Handling Operations}

-1341 :  BEGIN
         CASE  IR(15:17) OF
      0:     BEGIN                        {Compare character with storage: CCS}
         IF(AC(30 : 35) = M(Z;S,1:5)) THEN IC= IC+1
         IF(AC(30:35) < M(Z;S,1:5)) THEN IC = IC+2
         END

      1: BEGIN            CCS
         IF(AC(30:35) = M(Z;6:11)) THEN IC=IC+1
         IF(AC(30:35) < M(Z;6:11)) THEN IC=IC+2
         END
```

```
          2 :{Same as above, with memory location bits  12:17}
          3 :{ "    "    "     "     "      "       "   18:23}
          4 :{ "    "    "     "     "      "       "   24:29}
          5 :{ "    "    "     "     "      "       "   30:35}
          6 : BEGIN            {Storage minus test : MIT}
              IF (¬M(Z;S)) THEN  IC=IC+1
              END

          7 : BEGIN            {Storage plus test : PLT}
              IF( M(Z;S))  THEN IC= IC+1
              END
          CASEND
          END
 - -
-1623  :   BEGIN
             CASE  IR(15:17) OF
           0 : BEGIN            {Store Accumulator Character: SAC}
               M(Z;S,1:5) = AC(30:35)
               END
           1 : BEGIN            {SAC}
               M(Z;6:11) = AC(30:35)
               END
           2 :{Same as above with memory location bits  12 : 17}
           3 :{ "    "    "     "     "       "      "  18 : 23}
           4 :{ "    "    "     "     "       "      "  24 : 29}
           5 :{ "    "    "     "     "       "      "  30 : 35}

           6 : BEGIN          {Make storage sign minus : MSM}
               M(Z;S)=1
               END
           7 : BEGIN          {Make storage sign plus : MSP}
               M(Z;S)= 0
               END
           CASEND
           END
                              {Place character from storage: PCS}
 -1505  :   BEGIN
            CASE  IR(15:17) OF
            0 : BEGIN
                AC(30:35) = M(Z;S,1:5)
                END
            1 : BEGIN
                AC(30:35) = M(Z;6:11)
                END
```

```
        2  : {Same as above with memory location bits  12:17}
        3  : {"    "    "     "     "       "       "   18:23}
        4  : {"    "    "     "     "       "       "   24:29}
        5  : {"    "    "     "     "       "       "   30:35}
        CASEND
        END
```

{Data Transmission Instructions}

```
-1704 :   BEGIN                    {Transmit : TMT}
          Z=AC(3:17)
          ZD=AC(21:35)
          REPEAT BEGIN
                 M(ZD)= M(Z)
                 ZD=ZD+1
                 Z= Z+1
                  ⊥ IR
                 END
          UNTIL (IR(28:35) =0)
          END.
```

{Floating Point Instructions}

```
+0300 :   BEGIN                    { Floating point Add : FAD }
          IF(M(Z;S,1:8) > AC(S,1:8)) THEN
                          REPEAT BEGIN
                                  ⊥→ AC(9:35)°MQ
                                  ⊥ AC(S,1:8)
                                 END
                          UNTIL (M(Z;S,1:8)= AC(S,1:8))
          IF(M(Z;S,1:8) < AC(S,1:8)) THEN
                              REPEAT BEGIN
                                      ⊥→ M(Z;9:35)°MQ
                                      ↑ M(Z;S,1:8)
                                     END
                              UNTIL(M(Z;S,1:8)=AC(S,1:8))
          AC(9:35) = AC(9:35)+ M(Z;9:35)
          REPEAT BEGIN                      { Normalise }
                ←⊥ AC(9:35)°MQ
                 ⊥ AC(S,1:8)
                END
          UNTIL  (AC(1)≠ 1)
          END
```

```
-0300 : BEGIN                          {Unnormalised Floating Add: UFA}
        {Same as FAD without normalising the result}
        END

+0302 : BEGIN                         {Floating subtract : FSB}
        IF(M(Z;S,1:8)> A(S,1:8)) THEN
                              REPEAT  BEGIN
                                      ⊥→AC(9:35)°MQ
                                      ↑AC(S,1:8)
                                      END
                              UNTIL (M(Z;S,1:8)=AC(S,1:8))
        IF(M(Z;S,1:8) < AC(S,1:8)) THEN
                              REPEAT  BEGIN
                                      ⊥→M(Z;9:35)°MQ
                                      ↑M(Z;S,1:8)
                                      END
                            - UNTIL (M(Z;S,1:8)=AC(S,1:8))
        AC(9:35)=AC(9:35)- M(Z;9:35)
        REPEAT  BEGIN                  { Normalise }
                ←⊥AC(9:35)°MQ
                ↓AC(S,1:8)
                END
        UNTIL (AC(1)≠1)
        END

-302  : BEGIN                         {Unnormalised floating subtract:UFS}
        {Same as FSB without normalising the result}
        END

+0241:  BEGIN          -              {Floating divide or proceed: FDP}
        IF(M(Z;9:35) ≠0) THEN BEGIN
                                 AC(9:35)=AC(9:35)/M(Z;9:35)
                                 AC(S,1:8)=AC(S,1:8)- M(Z;S,1:8)
                                 END
        END

+0260 : BEGIN                         {Floating Multiply : FMP}
        AC(9:35)°MQ = AC(9:35)* M(Z;9:35)
        AC(S,1:8) = AC(S,1:8)+ M(Z;S,1:8)
        REPEAT  BEGIN                          { Normalise }
                ←⊥ AC(9:35)°MQ
                ↓AC(S,1:8)
                END
        END
```

```
-0260  :  BEGIN                  {Unnormalised Floating Multiply: UFM}
         {Same as FMP without normalising the result}
          END

+0301  :  BEGIN                 {Double presision floating add : DFAD}
         {Same as FAD but with two consecutive words
          M(Z) & M(Z+1) with AC  and MQ.
          Here sign bit is bit 'S' of Accumulator or M(Z).
          EXPONENT = AC(1:8) or M(Z;1:8) and
          MANTISA  = AC(9:35)°MQ(9:35) or
                        M(Z;9:35)°M(Z+1; 9:35)}
          END

+0303  :  BEGIN                  {Double presision floating subtract:DFSB}
         {Same as FSB with above changes in DFAD}
          END

+0261  :  BEGIN                 {Double precision Floating Multiply:DFMP}
         {Same as FMP with above changes in DFAD}
          END

-0241  :  BEGIN                  {Double Precision Floating Divide or
                                                     Proceed : DFDP}
         {Same as FDP with above changes in DFAD}
          END
          CASEND
          END

{Statement Part}

          BEGIN
          IF(RUN) THEN  BEGIN
                        IR=M(IC)
                        IC=IC+1
                        INSTRXEQ
                        END

          END
```

- * -